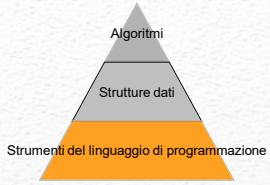
INTRODUZIONE E FONDAMENTI

«**Programmazione in C**», Kim N. King, Apogeo, Capitolo 1 e 2

«**C Primer Plus**», Stephen Prata, Addison Wesley, Cap. 1: Getting ready

Il Linguaggio C



Il Linguaggio di Programmazione C è costituito da

1. un linguaggio di computazione (il C vero e proprio)
2. un linguaggio di coordinazione fornito come "allegato", sotto forma di librerie standard:
   - alcune istruzioni "complesse" del linguaggio potrebbero essere realizzate in realtà da "mini-programmi" forniti insieme al compilatore, che li utilizza incorporandoli quando occorre.
   - LIBRERIE DI SISTEMA: insieme di componenti software che consentono di interfacciarsi col **sistema operativo**, usare le risorse da esso gestite, e realizzare alcune "istruzioni complesse" del linguaggio

C is remarkably **flexible**. It has been used for developing just about everything you can imagine by way of a computer program, from accounting applications to word processing and from games to operating systems. It is not only the basis for more advanced languages, such as C++, it is also used currently for developing mobile phone apps in the form of Objective C.

Il **sistema operativo** è un programma di controllo che svolge operazioni fondamentali, risiede in una memoria interna permanente e interpreta i comandi utente che richiedono varie specie di servizi, come la visualizzazione, la stampa o la copia di un file, il raggruppamento logico dei file in una directory o l'esecuzione di un programma. Il sistema operativo si occupa di gestire tutte le periferiche del calcolatore, tutti i processi, i dati di input/output.

Origini

Fu inizialmente creato nei laboratori della AT&T Bell Laboratories (anni ’70) per lo **sviluppo dei sistemi operativi** da Ken Thompson, Dennis Ritchie ed altri.

UNIX scritto in **assembly** (programmi faticosi da gestire, migliorare, …)

Thompson creò un piccolo linguaggio (B) per un ulteriore sviluppo di UNIX (basato su BCPL, a sua volta basato su Algol 60) e riscrisse una porzione di UNIX in B.

A partire dal ‘71 Ritchie iniziò lo sviluppo di una versione estesa del linguaggio (NB, NewB) che diventò C, stabile dal 1973, tanto che UNIX venne riscritto in C.

Importante beneficio: la **portabiltà**, scrivendo compilatori C per altri computer presenti nei laboratori Bell, il team potè far funzionare UNIX anche su tutte quelle macchine.

Assembly

L'assembly ha lo scopo generale di consentire al programmatore di ignorare il formato binario del linguaggio macchina. Ogni codice operativo del linguaggio macchina viene sostituito, nell'assembly, da una sequenza di caratteri che lo rappresenta in forma *mnemonica*; per esempio, il codice operativo per la somma potrebbe essere trascritto come ADD e quello per il salto come JMP

Il programma assembly risulta in questo modo relativamente più leggibile di quello in linguaggio macchina, con il quale mantiene però un totale (o quasi totale) isomorfismo. Il programma scritto in assembly non può essere eseguito direttamente dal processore; esso deve essere tradotto nel linguaggio macchina (binario) corrispondente, usando un programma compilatore detto assembler.

A causa di questa "vicinanza" all'hardware, non esiste un unico linguaggio assembly. Al contrario, ogni CPU o famiglia di CPU ha un proprio assembly, diverso dagli altri.

Assembly

Conoscere un certo linguaggio assembly implica saper scrivere programmi solo su una determinata CPU o famiglia di CPU. Passare ad altre CPU è relativamente facile, perché molti meccanismi sono analoghi o del tutto identici, quindi spesso il passaggio si limita all'apprendimento di nuovi codici mnemonici, nuove modalità di indirizzamento ed altre varie peculiarità del nuovo processore.

Molto meno facile è invece portare un programma scritto in assembly su macchine con processori diversi o con architetture diverse: quasi sempre significa dover riscrivere il programma da cima a fondo, perché i linguaggi assembly dipendono completamente dalla piattaforma per cui sono stati scritti.

A fronte di questi svantaggi **l'assembly offre un'efficienza senza pari e il controllo completo e assoluto sull'hardware**: i programmi in assembly sono, in linea di principio, i più **piccoli e veloci** che sia possibile scrivere su una data macchina.

Standardizzazione

Nel 1978 venne pubblicato il primo libro sul C "The C Programming Language", Kernigan, Ritchie, che divenne lo standard *de facto* (**K&R C**)

Nel 1980 C si era espanso ben oltre il mondo UNIX con compilatori disponibili su grande varietà di calcolatori con sistemi operativi differenti

Ma i programmatori che scrivevano nuovi compilatori si basavano su K&R, approssimativo su alcune caratteristiche del linguaggio

Inoltre il C continuò a cambiare anche dopo la pubblicazione del K&R

Necessità: descrizione del linguaggio **precisa**, **accurata** e **aggiornata** (**standard**) senza la quale i numerosi dialetti avrebbero minacciato la **portabilità** dei programmi C

Venne messo a punto uno standard statunitense chiamato ANSI C (American National Standards Institute) approvato nel 1989, poi approvato anche dell'ISO (International Organization for Standardization) nel 1990 (**C89 o C90**)

Nuovi cambiamenti vennero apportati nel 1999 (**C99**) con la pubblicazione del nuovo standard ISO

Standardizzazione

Il C99 non è ancora universalmente diffuso e ci vorranno anni affinchè tutti i compilatori diventino *C99-compliant*. Per un uso approfondito del linguaggio servirà conoscere le differenze fra C89 e C99. Ad esempio:

- **//commenti:** il C99 aggiunge secondo tipo di commenti
- **Identificatori**: C89 primi 31 caratteri significativi per gli identificatori, C99 63 caratteri
- **Valori restituiri dal main**: C89 funzione senza return valore restituito al s.o. indefinito, C99 se main dichiarato che restituisce int allora valore restituito al s.o. è 0
- **Dichiarazioni variabili**: C99 include la possibilità di dichiarare una variabile in un punto qualsiasi prima del suo utilizzo all'interno di un blocco (dichiarazioni e istruzioni anche mischiate)
- **Tipo restituito da funzione**: C99 non ammette l'omissione del tipo restituito da una funzione
- **Header <complex.h>**: C99 introduce <complex.h> che fornisce funzioni per eseguire operazioni matematiche sui numeri complessi

Lo standard più recente è definito dal document ISO/IEC 9899:2011 (**C11**)

Per un uso pienamente soddisfacente del C nell'implementazione di algoritmi le caratteristiche aggiunte al linguaggio con gli standard successivi non sono fondamentali

Linguaggi basati sul C

Enorme influenza sui linguaggi di programmazione moderni:

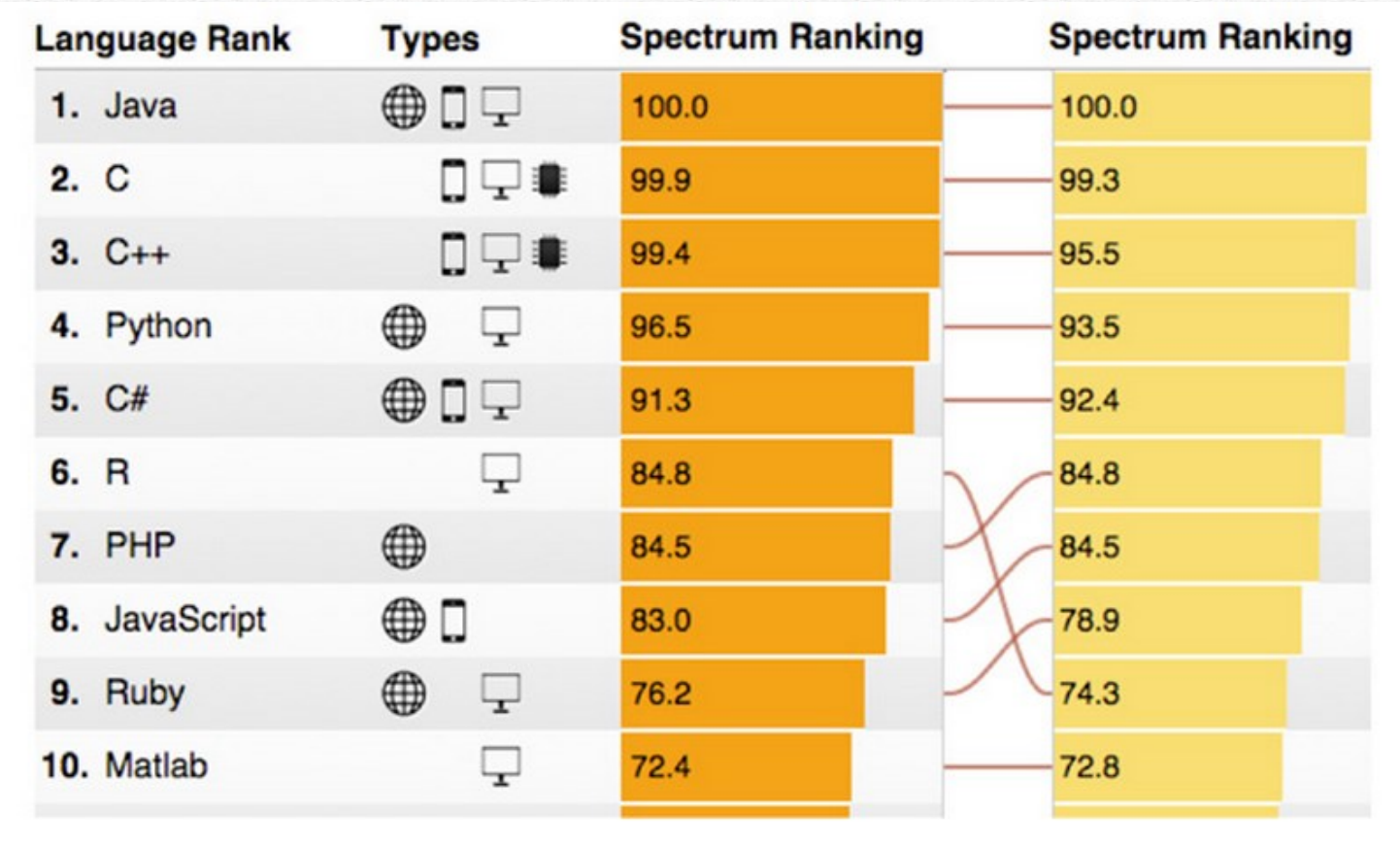
- **C++:** include tutte le caratteristiche del C e aggiunge il supporto alla programmazione orientata agli oggetti
- **Java**: basato sul C++ eredita molte delle caratteristiche del C
- **C#**: un più recente linguaggio basato su C++ e Java
- **Perl**: linguaggio di scripting che adotta molte delle caratteristiche del C

Linguaggio basilare per lo sviluppo di qualsiasi applicazione specialmente se la **memoria** o la **potenza di calcolo** sono limitate e **cruciali**, compresa l'implementazione di architetture operative (il Kernel di Linux è in gran parte scritto in tale linguaggio), di server web ed application server quali Apache, Tomcat, Websphere, IIS

Despite the popularity of newer languages, such as C++ and Java, C remains a core skill in the software business, typically ranking in the top 10 of desired skills.

The 2015 Top Ten Programming Languages

The ranking system is driven by weighting and combining 12 metrics from 10 data sources. The weighting of these sources can be adjusted using the interactive Web app to give, say, more importance to languages that have turned up in job ads.



http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages

The 2017 Top Ten Programming Languages

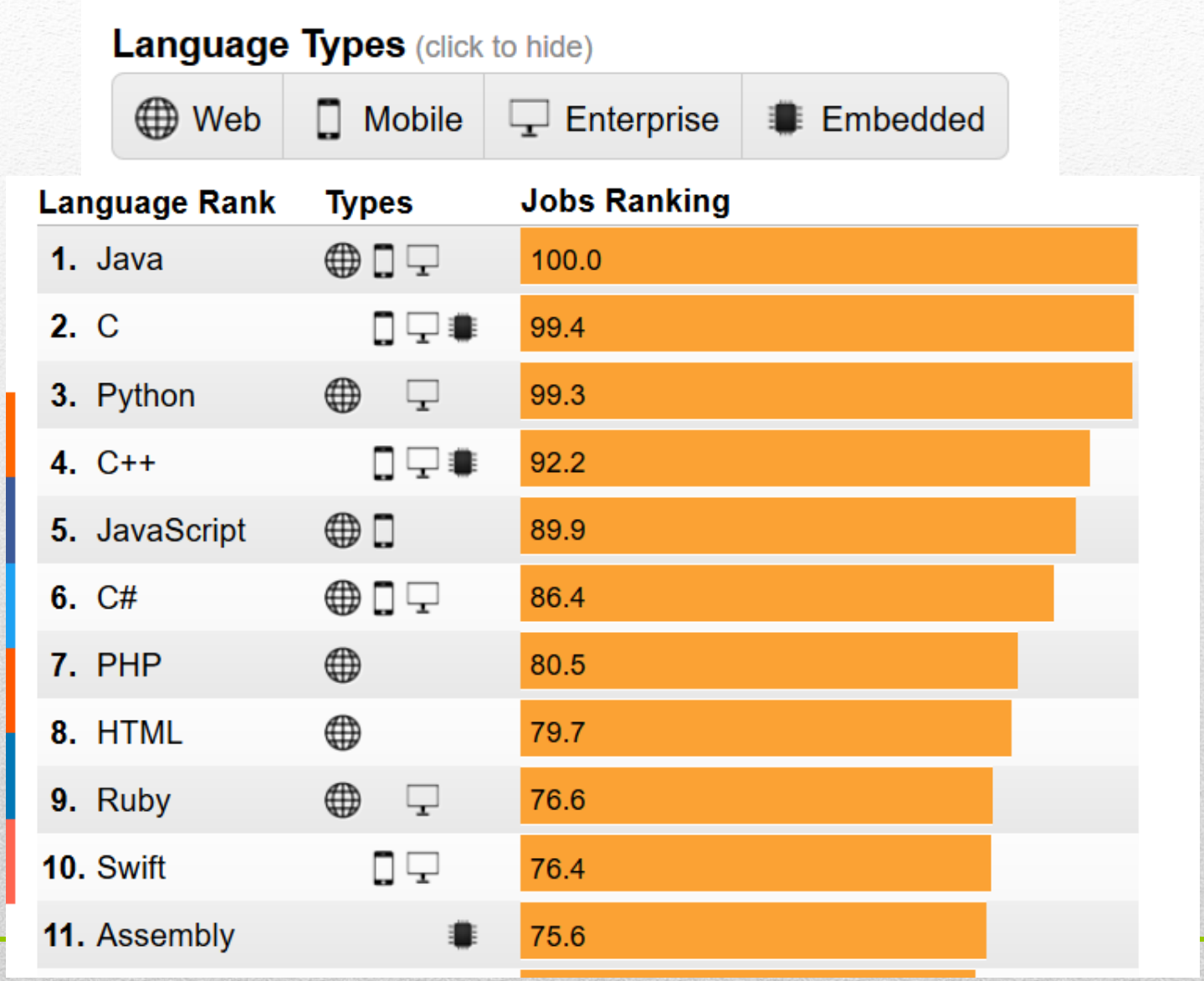
Language Types (click to hide)

Web | Mobile | Enterprise | Embedded

Language Rank	Types	Spectrum Ranking
1. Python	Web, Enterprise	100.0
2. C	Mobile, Enterprise, Embedded	99.7
3. Java	Web, Mobile, Enterprise	99.5
4. C++	Mobile, Enterprise, Embedded	97.1
5. C#	Web, Mobile, Enterprise	87.7
6. R	Enterprise	87.7
7. JavaScript	Web, Mobile	85.6
8. PHP	Web	81.2
9. Go	Web, Enterprise	75.1
10. Swift	Mobile, Enterprise	73.7

http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages

The 2017 Top Ten Programming Languages



Pregi

- **Efficienza:** pensato per applicazioni dove tradizionalmente veniva usato il linguaggio assembly, era cruciale che i programmi in C potessero girare velocemente e con una quantità di memoria limitata; ad es. include la gestione dei **puntatori**, fornendo al programmatore una **visibilità molto elevata sulla memoria** della macchina
- **Tipizzato**: debole controllo sui tipi di dato; a differenza di altri linguaggi il C permette di operare con assegnamenti e confronti su dati di tipo diverso, in qualche caso solo mediante una conversione di tipo esplicita (cast). Il compilatore demanda al programmatore il compito di verificare la correttezza semantica delle espressioni e la gestione di eventuali errori generati da espressioni non corrette (**flessibilità**)
- **Basso livello**: più vicino al linguaggio macchina con istruzioni basilari che vengono elaborate direttamente dal processore e permettono un totale accesso alle risorse della macchina; produce quindi un **codice più compatto ed efficiente**
- **Portabilità**: Consente lo sviluppo di **programmi facilmente portabili** da una piattaforma ad un'altra; la normativa ANSI stabilisce le caratteristiche standard di un compilatore C e la modalità standard di programmazione in C
- Disponibilità di **librerie standard** che contengono centinaia di funzioni deputate all'i/o, alla manipolazione delle stringhe, alla gestione della memorizzazione, …

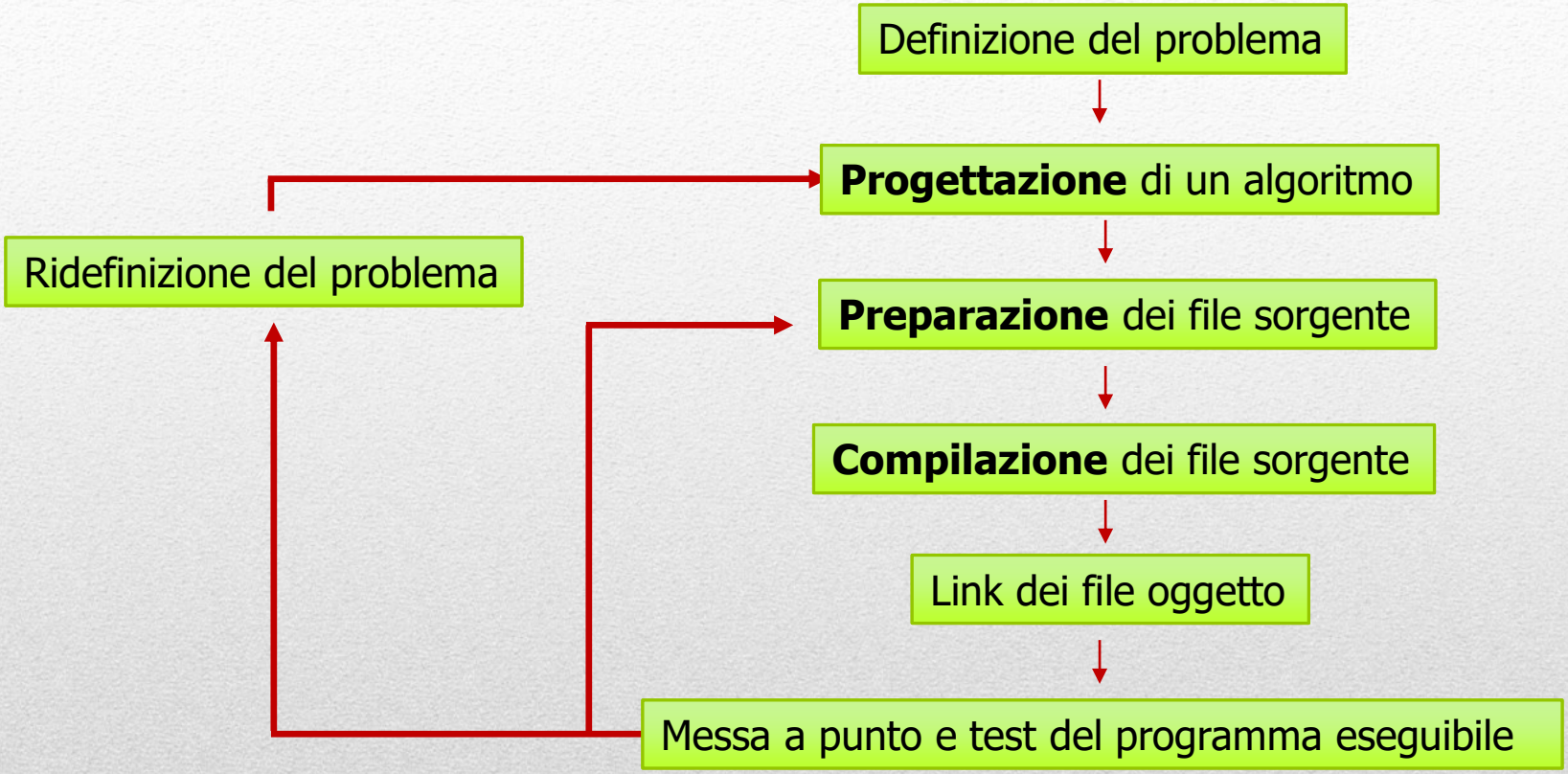
Debolezze

- **Programmi inclini agli errori** a causa della flessibilità del C e delle possibili difficoltà per programmatori non accorti (provate a dimenticare un «&» ove necessario!)
- **Programmi difficili da capire** a causa della natura stringata e succinta dei programmi (linguaggio conciso) e della sua flessibilità se esasperata
- **Programmi difficili da modificare** se non sono stati sviluppati considerando la necessità di manutenzione del codice

In short, C is one of the most important programming languages and will continue to be so.

The International Obfuscated C Code Contest http://www.ioccc.org/

Lo sviluppo dei programmi



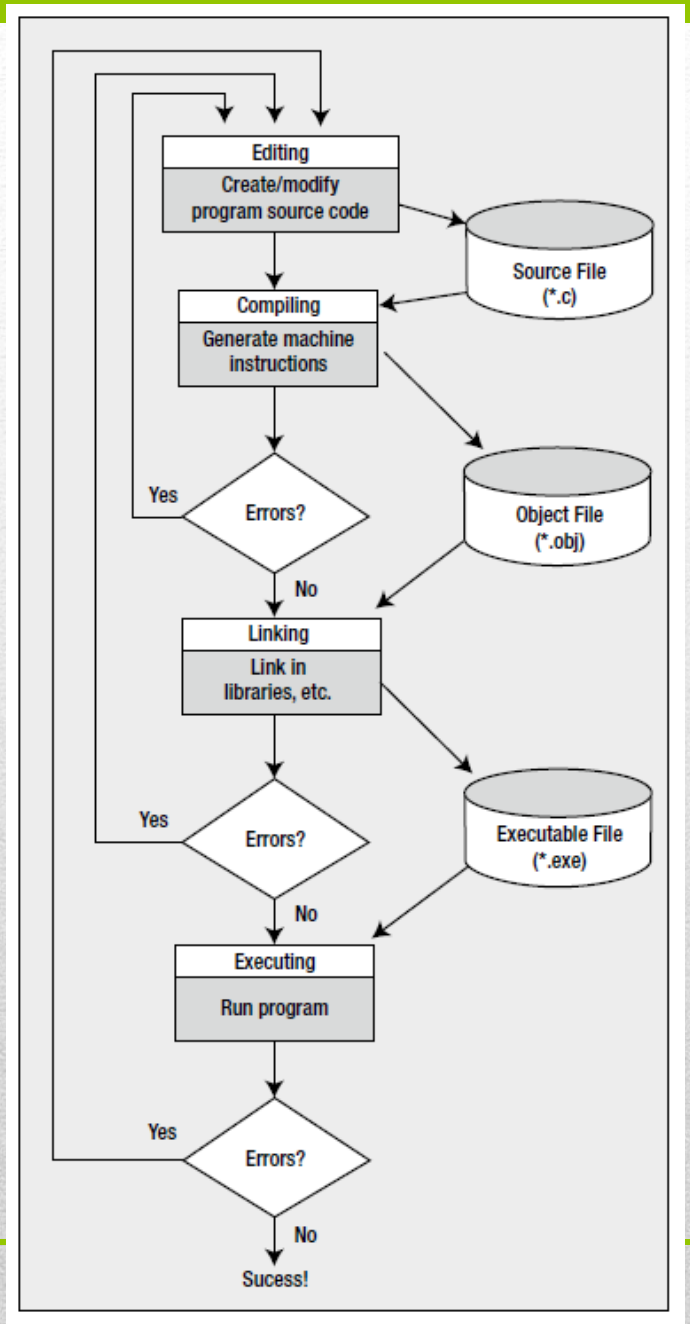
N.B.: Gli argomenti di inizio corso vengono al momento poco più che accennati, saranno tutti approfonditi nel seguito; ora abbiamo bisogno di porre le fondamenta dei diversi temi e di cominciare a lavorare sulla macchina

Lo sviluppo

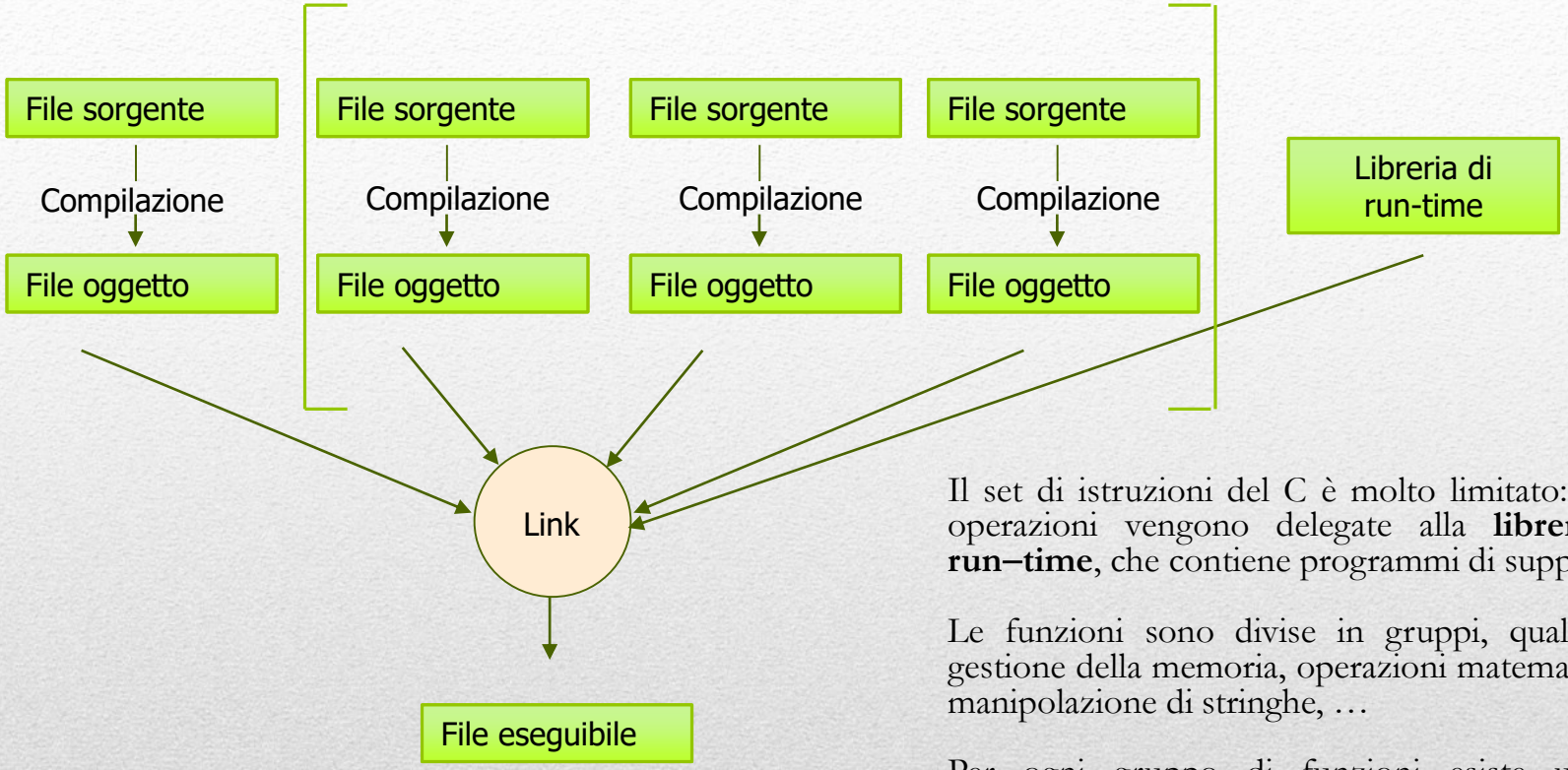
Creare ed eseguire un programma:

The execution stage is where you run your program, having completed all the previous processes **successfully**.
Unfortunately, this stage can also generate a wide variety of **error conditions** that can include producing the wrong output, just sitting there and doing nothing, or perhaps crashing your computer for good measure. In all cases, it's back to the editing process to check your source code.

Your program may be quite accurate from a language point of view, and it may compile and run without a problem, but it won't produce the right answers. These kinds of errors can be **the most difficult to find**.



Lo sviluppo dei programmi



Il set di istruzioni del C è molto limitato: molte operazioni vengono delegate alla **libreria di run–time**, che contiene programmi di supporto

Le funzioni sono divise in gruppi, quali I/O, gestione della memoria, operazioni matematiche e manipolazione di stringhe, …

Per ogni gruppo di funzioni esiste un file sorgente, chiamato ***file header*****,** contenente le informazioni necessarie per utilizzare le funzioni

I codici sorgente ed oggetto possono essere suddivisi in più file, il codice eseguibile di un programma risiede in un unico file

Object Code Files, Executable Files, and Libraries

In practice, a program of any significant size will consist of **several source code** files, from which the compiler generates object files that need to be linked. A large program may be **difficult to write in one working session**, and it may be impossible to work with as a single file. By breaking it up into a number of smaller source files that each provide a **coherent part** of what the complete program does, you can make the development of the program a lot easier. The source files can be **compiled separately**, which makes eliminating simple typographical errors a bit easier.

Compiling + linking: the compiler converts your source code to an intermediate code, and the linker combines this with other code to produce the executable file.

C uses this **two-part approach** to facilitate the **modularization of programs**. You can compile individual modules **separately** and then use the linker to combine the compiled modules later. That way, if you need to change one module, you don't have to recompile the other ones.

Also, the linker combines your program with precompiled library code.

Object Code Files, Executable Files, and Libraries

The object file contains the translation of your source code, but it is **not yet a complete program:**

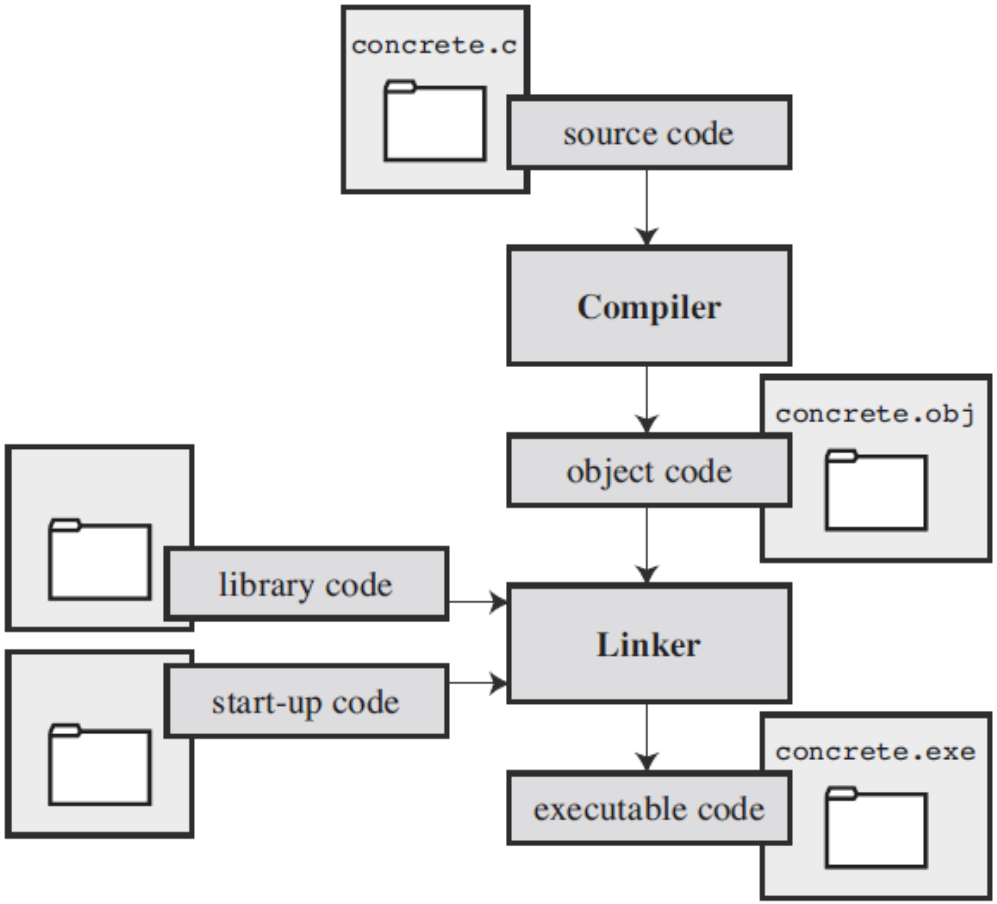
1. startup code, which is code that acts as an **interface** between your **program** and the **operating system**.

For example, you can run an IBM PC compatible under MS Windows or under Linux. The **hardware is the same** in either case, so the same object code would work with both, but you would **need different startup code** for Windows than you would for Linux because these systems handle programs differently from one another.

2. *code for library routines*, the object code file does not contain the code for this function; it merely contains instructions saying to use the *printf()* function. The actual code is stored in another file, called a *library*. A library file contains object code for many functions.

Compiler and linker

The role of the linker is to bring together these three elements: **your object code**, the **standard startup code** for your system, and the **library code**—and put them together into a single file, **the executable file**. For library code, the linker extracts only the code needed for the functions you use from the library.



In short, an object file and an executable file both consist of **machine language instructions**. However, the object file contains the machine language translation only for the code you used, but the executable file also has machine code for the library routines you use and for the startup code.

Un semplice programma in C: scrittura di una riga di testo

```
1
2  /*   A first program in C */
3  #include <stdio.h>
4
5  int main()
6  {
7     printf( "Welcome to C!\n" );
8
9     return 0;
10 }
```

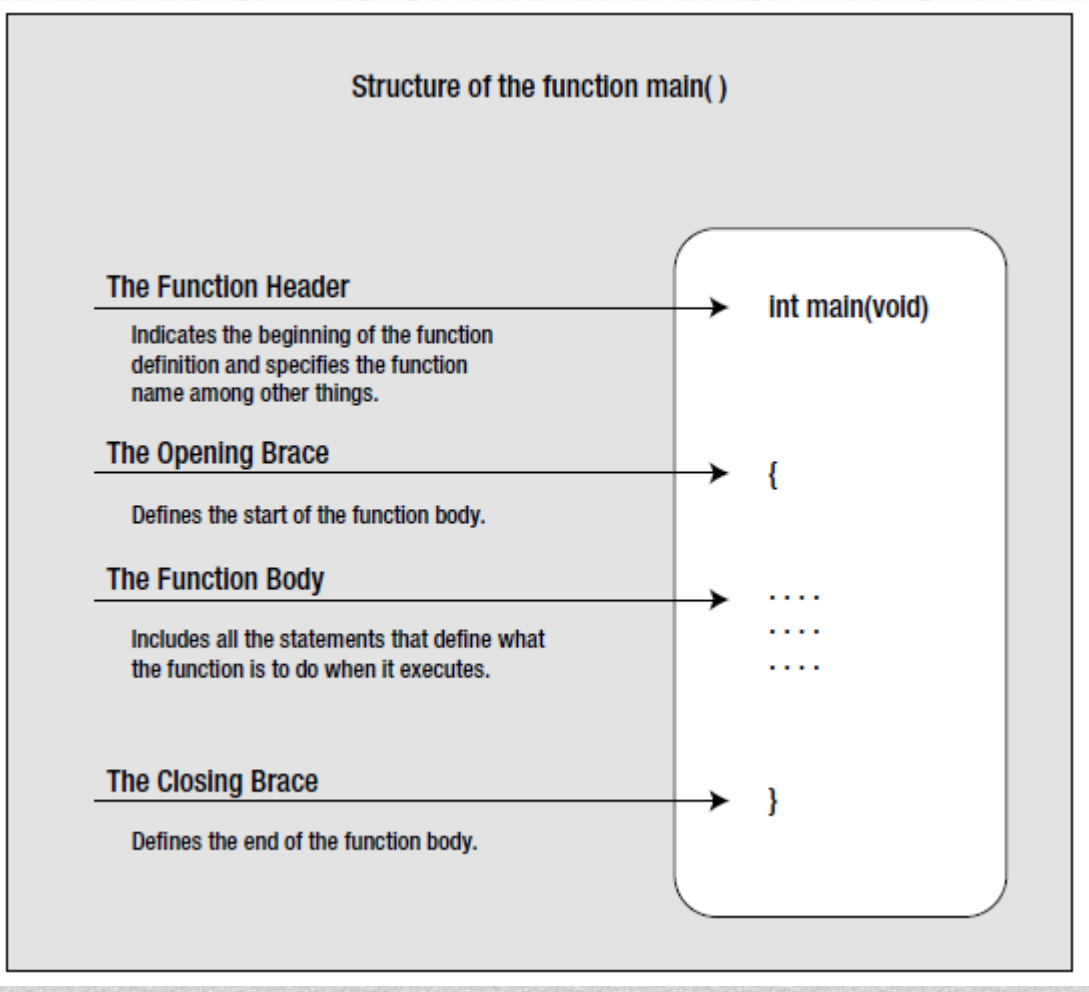
IL PROGRAMMA SORGENTE

```
Welcome to C!
```

IL RISULTATO

- Commenti
  - Il testo comprteso fra /* e */ è ignorato dall'elaboratore
  - E' utile per commentare e descrivere il programma
- #include <stdio.h>
  - Direttiva del preprocessore: dice all'elaboratore di caricare la libreria <stdio.h> che contiene le operazioni di input/output standard poichè il C non ha comandi incorporati di lettura/scrittura

Un semplice programma in C: scrittura di una riga di testo



Commenti al programma

- int main()
  - Una delle funzioni in tutti i programmi in C deve essere main
  - Le parentesi tonde indicano una funzione
  - int significa che main "restituisce" un valore di tipo intero. Il valore restituito dalla funzione main (tipicamente un intero) può essere utilizzato da un eventuale programma chiamante.
  - Fino a quando non impareremo a scrivere altre funzioni, il *main* sarà l'unica dei nostri programmi
- Le parentesi graffe indicano un blocco di programma
  - Il "corpo" delle funzioni deve essere racchiuso fra parentesi graffe

Funzione: dalla matematica (regola per calcolare un valore a partire da uno o più argomenti dati) indica un raggruppamento di istruzioni al quale è stato assegnato un nome. Alcune calcolano un valore, altre no. Nel primo caso usiamo l'istruzione *return* per specificare il valore che restituisce

Commenti al programma

- printf("Welcome to C!\n");
  - Indica l'esecuzione di un'azione
    - Stampa la stringa di caratteri compresa fra gli apici
  - Tutta la linea è detta istruzione
    - Tutte le istruzioni devono terminare con il ;
  - \ - carattere di escape
    - Indica che printf deve eseguire qualcosa di particolare
    - \n è il carattere di "a capo"
- return 0;
  - E' un modo di "uscire" da una funzione
  - return 0, in questo caso indica che il programma termina normalmente
- Parentesi graffa }
  - Indica che si è raggiunta la fine del main

Altri esempi

```
/*
 * File esempio1.c
 * Scopo: utilizzare funzione printf()
 * Autore: Mario Rossi
 * Data: 23 luglio 2015
*/

#include <stdio.h>
int main( )
{
   printf("Welcome to C!\n");
   return 0;
}
```

```
// File esempio2.c
// Scopo: stampare su un'unica riga
// con due funzioni printf()
// Autore: Mario Rossi
// Data: 23 luglio 2015

#include <stdio.h>
int main( )
{
   printf("Welcome ");
   printf("to C!\n");
   return 0;
}
```

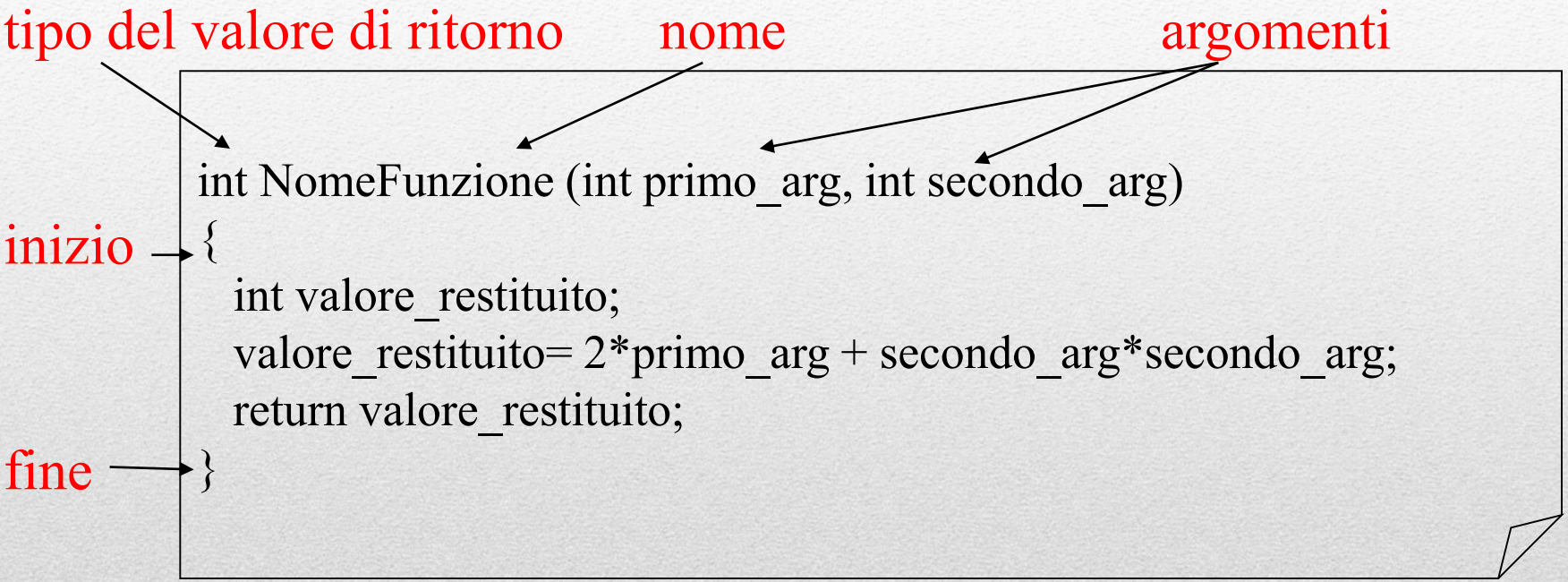
Machine-dependent facilities such as input and output for your computer are implemented by the standard library in a **machine-independent form**. This means that **you write data to a disk file in C in the same way on your PC as you would on any other kind of computer**, even though the underlying **hardware processes** are quite different.

Caratteristiche di ogni programma C

- La parentesi graffa aperta { indica l'inizio di un blocco di istruzioni
- La parentesi graffa chiusa } indica la fine di un blocco di istruzioni
- Per ogni { deve essercene una chiusa }
- I commenti possono essere posti ovunque e verranno ignorati dal compilatore
- /* questo è l'inizio del commento e questa la sua fine */
- Il C99 prevede un secondo tipo di commenti che iniziano con // e che terminano automaticamente alla fine della riga
- L'annidamento di commenti non è permesso
- Il C è case sensitive.
- Il linguaggio C è costituito di sole 32 parole chiave riservate che combinate realizzano la **sintassi formale** del linguaggio (scritte in minuscolo). Una parola chiave non può essere utilizzata per scopi differenti
- Il linguaggio di programmazione C offer un approccio strutturato alla programmazione

Generalizziamo

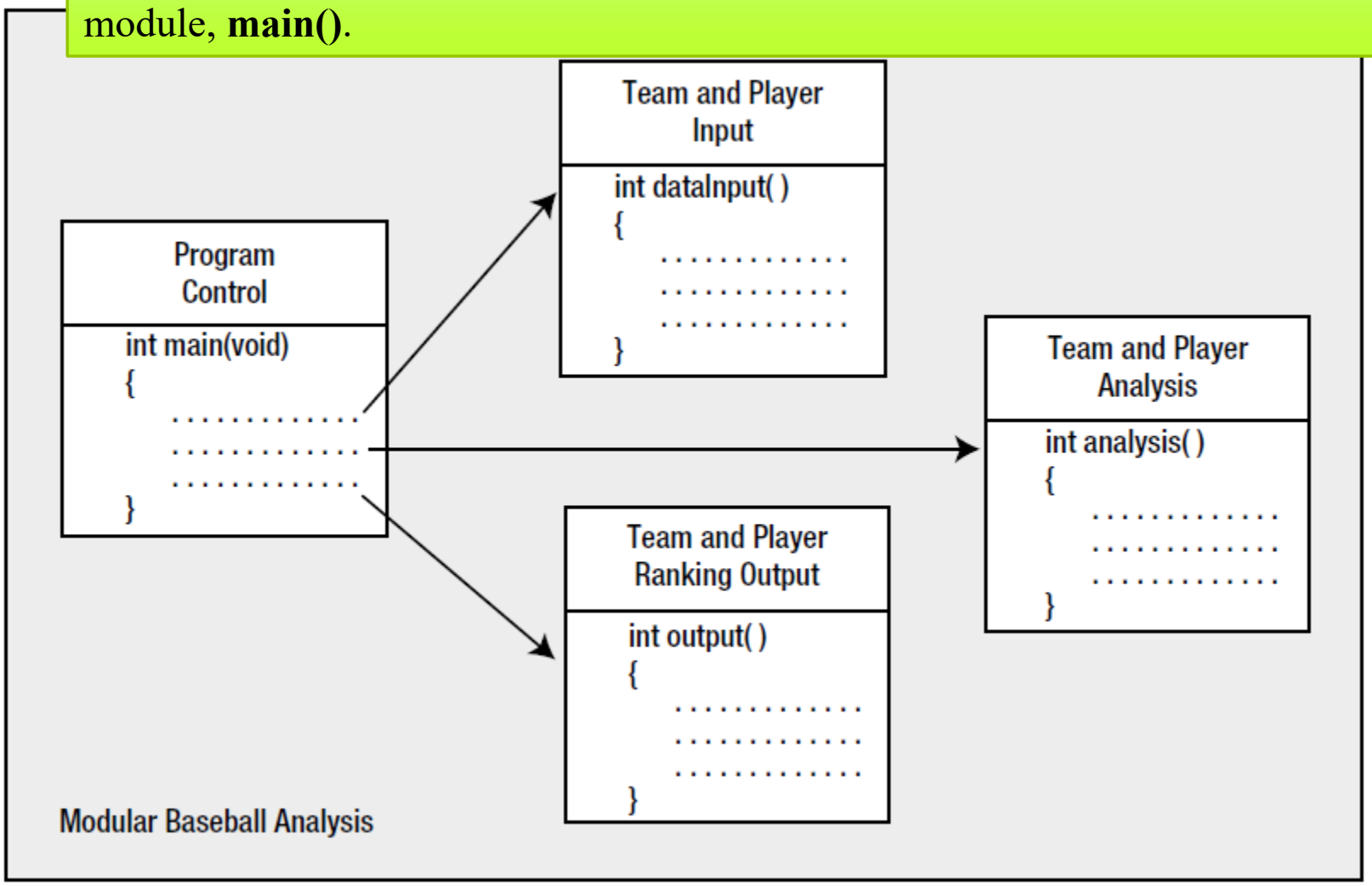
Unità fondamentale: ***funzione***



main: nome speciale di funzione dalla quale inizia l'esecuzione del programma

Generalizziamo

A simple example of a program to analyze baseball scores that is composed of four functions Each of the four functions does a specific, well-defined job. Overall control of the sequence of operations in the program is managed by one module, **main()**.



Generalizziamo

Each of the four functions does a **specific**, **well-defined** job. **Overall control** of the sequence of operations in the program is managed by **one module**, main().

There is a function to read and check the input data and another function to do the analysis. Once the data have been read and analyzed, a fourth function has the task of outputting the team and player rankings.

Generalizziamo

Segmenting a program into manageable chunks:

- It allows each function to be **written and tested separately**. This greatly simplifies the process of getting the total program to work.
- Several separate functions are **easier to handle and understand** than one huge function.
- Libraries are just sets of functions that people tend to use all the time. Because they've been prewritten and pretested, you know that they work, so **you can use them without worrying about their code details**. This will accelerate your program development by allowing you to concentrate on your own code, and it's a fundamental part of the philosophy of C. **The richness of the libraries greatly amplifies the power of the language**.

Generalizziamo

- You can accumulate **your own libraries of functions** that are applicable to the sort of programs that you're interested in. If you find yourself writing a particular function frequently, you can write a **generalized version of it** to suit your needs and build this into your own library. Then, whenever you need to use that particular function, you can simply use your library version.
- In the development of large programs, which can vary from a few thousand to millions of lines of code, development can be undertaken by **teams of programmers**, with each team working with a defined subgroup of the functions that make up the whole program.

La struttura di un programma C

```
direttive per il preprocessore
dichiarazioni globali
main( )
{
  variabili locali alla funzione main ;
  istruzioni della funzione main
}
f1( )
{
  variabili locali alla funzione f1 ;
  istruzioni della funzione f1
}
f2( )
{
  variabili locali alla funzione f2 ;
  istruzioni della funzione f2
}
 . . .
 etc
```

Anche il più semplice programma C si basa su tre **componenti chiave** del linguaggio:

- le **direttive**: modificano il programma prima della compilazione; iniziano con il carattere # e **non** terminano con il ;
- le **funzioni**: ad es. il *main*
- le **istruzioni**: comandi eseguiti durante l'esecuzione del programma

Si noti l'utilizzo delle parentesi tonde e graffe.

Le parentesi tonde ( ) vengono utilizzate in unione con i nomi delle funzioni, mentre le parentesi graffe {} vengono utilizzate per delimitare le espressioni.

Infine il punto e virgola ; indica la terminazione di un'espressione in C.

Il linguaggio C ha un formato piuttosto flessibile: espressioni lunghe possono essere continuate su righe successive senza problemi: il punto e virgola segnala al compilatore che è stata raggiunta la fine dell'espressione.

Inoltre, è possibile introdurre spazi indiscriminatamente, in genere allo scopo di dare un miglior aspetto al programma.

Un errore molto comune è dimenticarsi il punto e virgola: in questo caso il compilatore concatenerà più linee di codice generando un'espressione priva di qualsiasi significato. Per questo motivo il messaggio d'errore che il compilatore restituisce non è la mancanza del punto e virgola, quanto la presenza di un qualcosa di incomprensibile. Attenzione ad interpretare i messaggi del compilatore.

Le direttive al preprocessore

Una direttiva al preprocessore *(#include)* permette di inserire tutto il testo del file specificato nel nostro sorgente, a partire dalla riga in cui si trova la direttiva stessa.

In particolare, in *stdio.h* è descritto il modo in cui la funzione printf() si **interfaccia** al programma che la utilizza; ci sono poi altre direttive al preprocessore e definizioni che servono al compilatore per tradurre correttamente il programma.

Ogni compilatore C è accompagnato da un certo numero di file *.h*, detti **include file** o **header file**, il cui contenuto è necessario per un corretto utilizzo delle funzioni di libreria (anche le librerie sono fornite col compilatore).

Gli header contengono i "**prototipi**" (le dichiarazioni) delle funzioni di libreria e la definizione di costanti. La parola chiave "*include*" non è una istruzione del linguaggio C e quindi non deve essere terminata da *;*. Esempio:

```
#include <stdio.h>
#include <math.h>
```

Le librerie

Contengono codice che esegue funzioni quali ad es. le funzioni matematiche, le funzioni di I/O, le funzioni grafiche, …

The standard functionality that the library contains includes capabilities that most programmers are likely to need, such as processing text strings or math calculations. This **saves you an enormous amount of effort** that would be required to implement such things yourself.

In pratica tutte le **operazioni di interazione** tra i programmi e l'hardware, il firmware ed il sistema operativo sono delegate a funzioni (aventi interfaccia più o meno standardizzata) esterne al compilatore, il quale non deve dunque implementare particolari capacità di generazione di codice, peculiari per il sistema al quale è destinato, permettendo la sua **portabilità**.

Molti compilatori mettono a disposizione librerie proprietarie, che facilitano la programmazione ma che rendono difficile il porting del programma da un'architettura ad una differente.

I cosiddetti "ambienti di sviluppo integrati", quali ad es. Microsoft Visual C, generalmente facilitano la soluzione delle problematiche relative alle librerie da usare, così come nascondono la maggior parte degli aspetti della compilazione, mediante un'interfaccia visuale, e facilitano le fasi di debugging, permettendo di seguire passo passo l'esecuzione del programma.

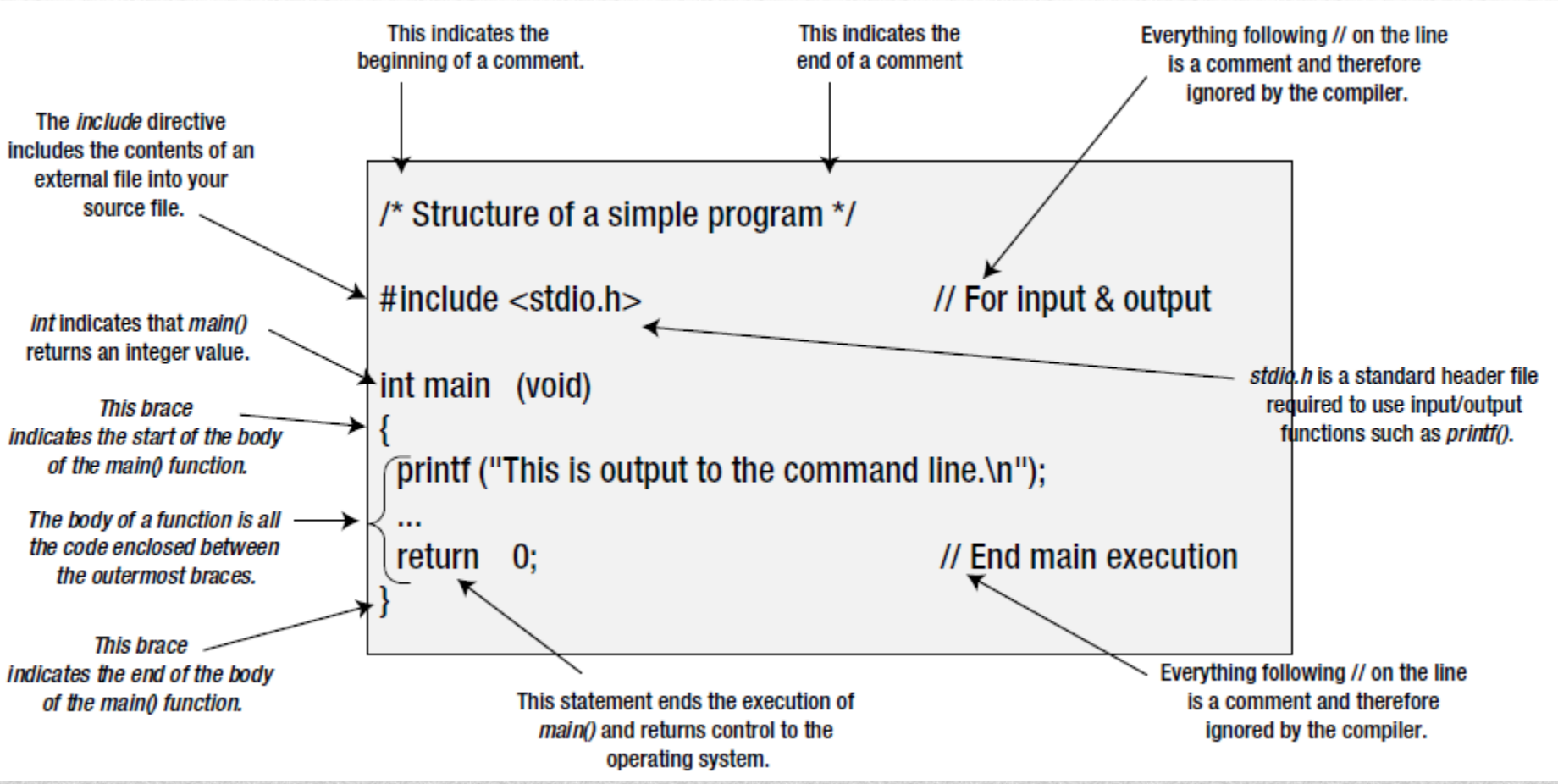
Le librerie standard

Per usare una libreria, non occorre inserirla esplicitamente nel progetto: ogni ambiente di sviluppo sa già dove cercarle

Ogni file sorgente che ne faccia uso deve però includere lo header opportuno, che contiene le dichiarazioni necessarie.

•input/output	stdio.h
•funzioni matematiche	math.h
•gestione di stringhe	string.h
•operazioni su caratteri	ctype.h
•gestione dinamica della memoria	stdlib.h
•operazioni su data e ora	time.h
•…	
•... e molte altre.	

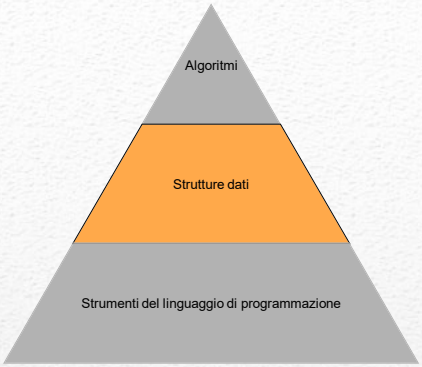
In conclusione



RISOLUZIONE DI UN PROBLEMA

«Algoritmi e strutture dati»
Camil Demetrescu, Irene Finocchi, Giuseppe F. Italiano, McGraw Hill
Capitolo 3

Risoluzione di un problema



La risoluzione di un problema mediante la realizzazione di un programma richiede i seguenti passi:

1. Definizione di un modello che sintetizzi le caratteristiche essenziali del problema (l'algoritmo di soluzione a questo stadio può essere espresso solo in termini informali)
2. Definizione dell'**algoritmo** in uno pseudo-linguaggio, tramite r**affinamenti successiv**i fino a quando non vengono specificate in un certo dettaglio le operazioni che vanno compiute sui tipi di dati. Vengono cioè creati dei tipi di dato astratti (**abstract data type**) in cui ciascuna operazione sul tipo è una funzione con nome appropriato.
3. **Realizzazione dell'ADT** e delle **funzioni** che su di esso operano e delle restanti parti di algoritmo.

Risoluzione di un problema

Un mattone essenziale nella realizzazione di programmi è la procedura (**funzione**) che ha le seguenti caratteristiche:

- generalizza la nozione di operatore consentendo l'estensione di quelli forniti dal linguaggio (es. moltiplicazione di matrici)
- incapsula parti di un algoritmo localizzando in una sezione del programma tutte le istruzioni riguardanti aspetti comuni.

Risoluzione di un problema

Un ADT è invece un modello matematico insieme con la collezione delle operazioni definite su quel modello. Inoltre:

- generalizza i tipi primitivi (reale, intero, …) come una procedura fa con gli operatori sui tipi primitivi
- incapsula un tipo e le operazioni valide su di esso come una procedura fa con una parte di programma. Così i cambiamenti ad un ADT hanno impatto solo all'interno dell'ADT, l'interfaccia verso l'esterno resta identica e i dettagli non necessari sono invisibili dall'esterno.

Specifica *cosa* un'operazione (funzione) che agisce sul tipo di dato deve fare, ma non *come* l'operazione (funzione) può essere realizzata e soprattutto *come* gli oggetti della collezione possono essere organizzati in modo che le operazioni siano efficienti e la collezione stessa occupi poco spazio di memoria.

Esempio: tipo di dato Dizonario, Lista, Pila, Albero, …

Uso di ADT

L'uso di ADT generalizza il concetto di tipo in C, che offre:

- tipi primitivi su cui sono disponibili operazioni predefinite
- tipi definiti dall'utente sui quali non esistono operazioni predefinite.

Un ADT incapsula in un solo involucro una definizione di tipo e le operazioni su di esso, rendendo i tipi definiti dall'utente più omogenei a quelli predefiniti.

L'interfaccia lo caratterizza verso l'esterno.

L'implementazione è una traduzione in termini di linguaggio di programmazione della dichiarazione della variabile con quelle proprietà e di una funzione per ogni operazione dell'ADT. Una implementazione sceglie una particolare struttura dati per la rappresentazione del tipo, usando i tipi primitivi ed i metodi di strutturazione dati disponibili nel linguaggio scelto.

In generale, per uno stesso ADT sono possibili molteplici realizzazioni alternative basate su **strutture dati** diverse che permettono di implementare le operazioni richieste in modo più o meno efficiente.

Tipo e struttura dati

Il tipo di una variabile è l'insieme dei valori che essa può assumere.

Una struttura dati è la collezione di variabili di vari tipi connesse in vari modi (tipicamente per realizzare il modello matematico di un ADT).

I metodi usati per connettere variabili dei vari tipi e far loro formare una struttura dati sono legati al linguaggio di programmazione (es. array, record, file, struct, …), così come la possibilità di esprimere relazioni fra variabili (puntatori).

Gestione di collezioni di oggetti

Tipo di dato:

- Specifica delle operazioni di interesse su una collezione di oggetti (es. inserisci, cancella, cerca)

Struttura dati:

- Organizzazione dei dati che permette di supportare le operazioni di un tipo di dato usando meno risorse di calcolo possibile

Il tipo di dato Dizionario

tipo `Dizionario`:
dati:
un insieme S di coppie $(elem, chiave)$.

operazioni:

`insert`$(elem\ e, chiave\ k)$
aggiunge a S una nuova coppia (e, k).

`delete`$(chiave\ k)$
cancella da S la coppia con chiave k.

`search`$(chiave\ k) \rightarrow elem$
se la chiave k è presente in S restituisce l'elemento e ad essa associato, e `null` altrimenti.

Il tipo di dato Pila

tipo `Pila`:
dati:
una sequenza S di n elementi.

operazioni:

`isEmpty`$() \rightarrow result$
restituisce `true` se S è vuota, e `false` altrimenti.

`push`$(elem\ e)$
aggiunge e come ultimo elemento di S.

`pop`$() \rightarrow elem$
toglie da S l'ultimo elemento e lo restituisce.

`top`$() \rightarrow elem$
restituisce l'ultimo elemento di S (senza toglierlo da S).

Il tipo di dato Coda

tipo `Coda`:
dati:
una sequenza S di n elementi.

operazioni:

`isEmpty`$() \rightarrow result$
restituisce `true` se S è vuota, e `false` altrimenti.

`enqueue`$(elem\ e)$
aggiunge e come ultimo elemento di S.

`dequeue`$() \rightarrow elem$
toglie da S il primo elemento e lo restituisce.

`first`$() \rightarrow elem$
restituisce il primo elemento di S (senza toglierlo da S).

Tecniche di rappresentazione dei dati

Rappresentazioni indicizzate:

- I dati sono contenuti in array

Rappresentazioni collegate:

- I dati sono contenuti in record collegati fra loro mediante puntatori

Pro e contro

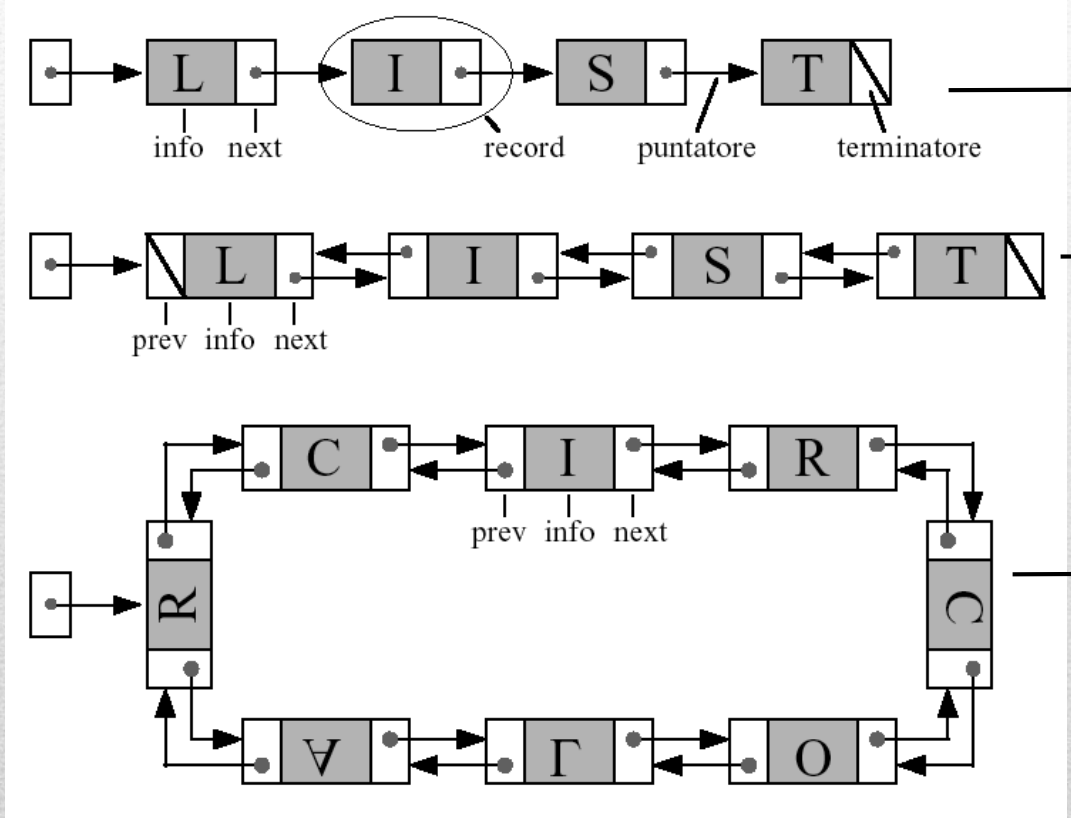
Rappresentazioni indicizzate:

- Pro: accesso diretto ai dati mediante indici
- Contro: dimensione fissa (riallocazione array richiede tempo lineare)

Rappresentazioni collegate:

- Pro: dimensione variabile (aggiunta e rimozione record in tempo costante)
- Contro: accesso sequenziale ai dati

Esempi di strutture collegate



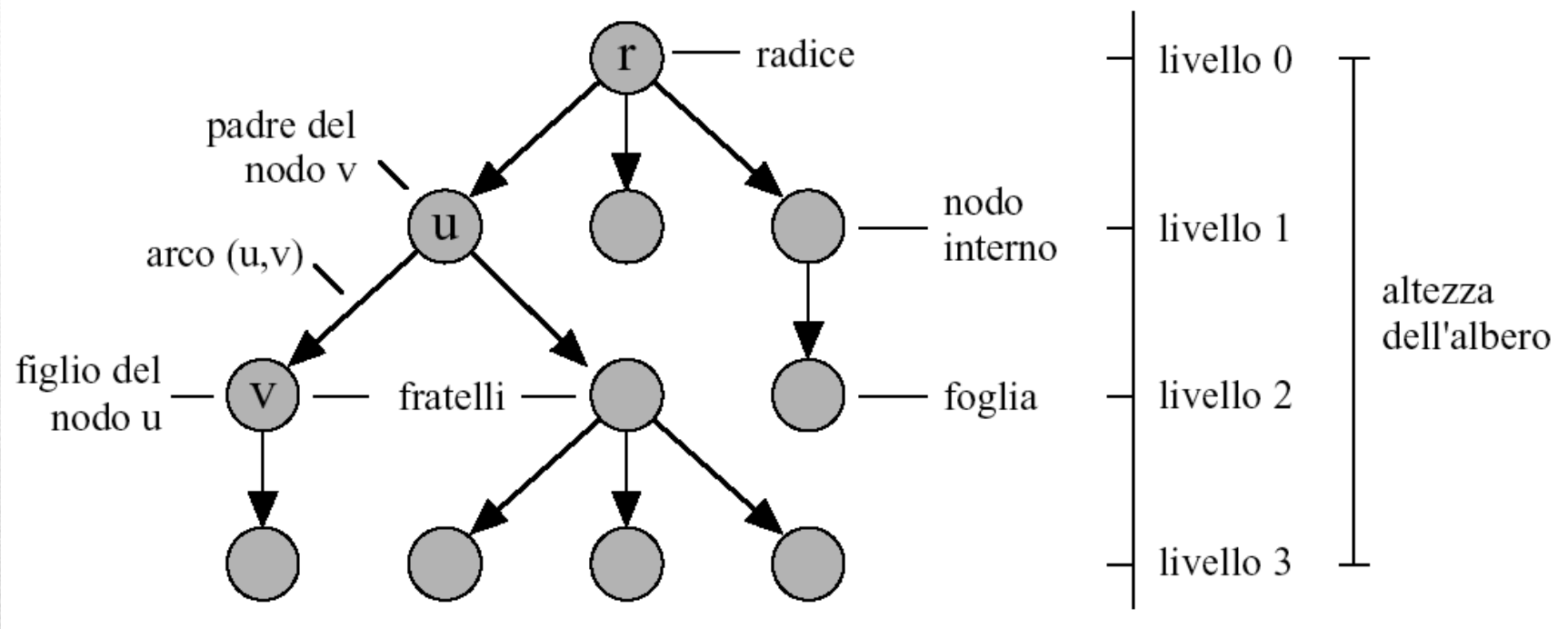
Lista semplice

Lista doppiamente collegata

Lista circolare doppiamente collegata

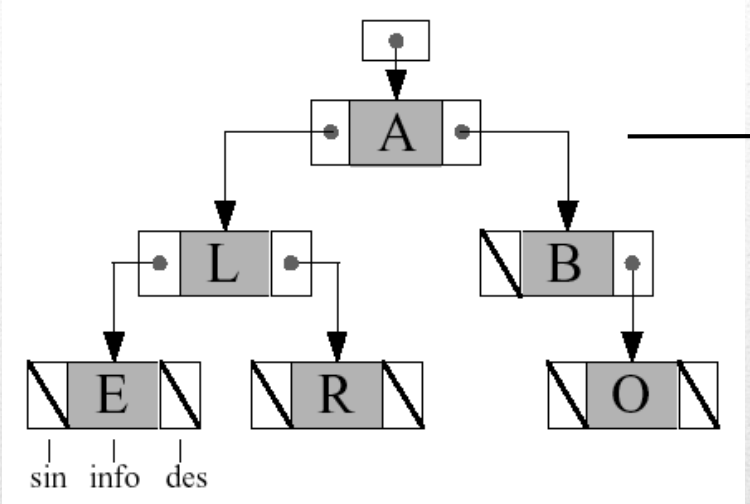
Alberi

Organizzazione gerarchica dei dati:



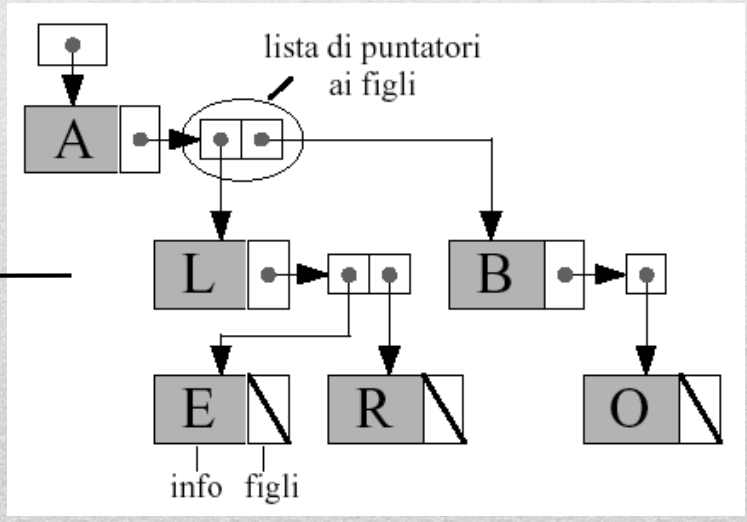
Dati contenuti nei nodi, relazioni gerarchiche definite dagli archi che li collegano.

Rappresentazioni collegate di alberi

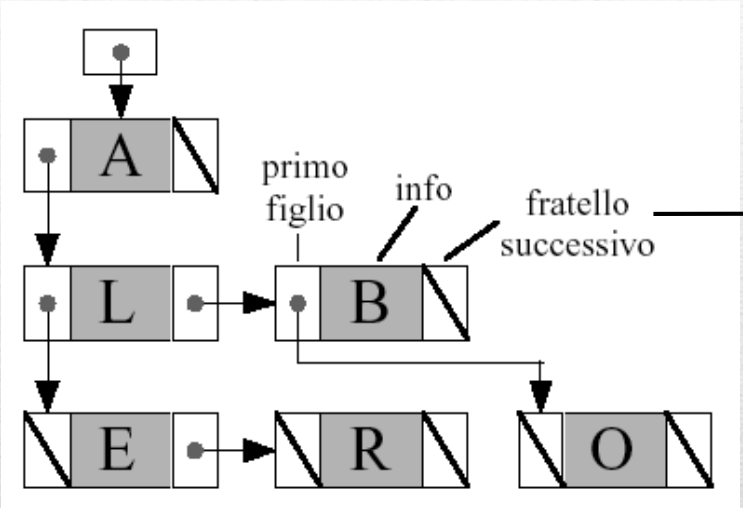


Rappresentazione con puntatori ai figli (nodi con numero limitato di figli)

Rappresentazione con liste di puntatori ai figli (nodi con numero arbitrario di figli)



Rappresentazioni collegate di alberi



Rappresentazione
di tipo primo figlio-fratello successivo (nodi con numero arbitrario di figli)